Component-based Programming for Higher-Order Attribute Grammars

João Saraiva

Department of Computer Science,
University of Minho, Braga, Portugal
jas@di.uminho.pt

Abstract. This paper presents techniques for a component-based style of programming in the context of higher-oder attribute grammars (HAG). Attribute grammar components are "*plugged in*" into larger attribute grammar systems through higher-order attribute grammars. Higher-order attributes are used as (intermediate) "gluing" data structures.
This paper also presents two attribute grammar components that can be re-used across different language-based tool specifications: a visualizer and animator of programs and a graphical user interface AG component. Both components are reused in the definition of a simple language processor. The techniques presented in this paper are implemented in Lrc: a purely functional, higher-order attribute grammar-based system that generates language-based tools.

1 Introduction

Recent developments in programming languages are changing the way we construct programs. Programs are now a collection of generic, reusable, off-the-shelf program components that are "*plugged in*" to form larger and powerful programs. In such an architecture, intermediate gluing data structures are used to convey information between different program components: a component constructs (produces) an intermediate data structure which is used (consumed) by other component.

In the context of the design and implementation of language-based tools, attribute grammars provide powerful properties to improve the productivity of their users, namely, the static scheduling of computations. Indeed, an attribute grammar writer is neither concerned with breaking up her/his algorithm into different traversal functions, nor is she/he concerned in conveying information between traversal functions (*i.e.*, how to pass intermediate values computed in one traversal function and used in following ones). A second important property is that circularities are statically detected. Thus, the existence of cycles, and, as a result, the non-termination of the algorithms, is detected statically. That is to say that for (ordered) attribute grammars the termination of the programs for all possible inputs is statically guaranteed. A third characteristic is that attribute grammars are declarative. Furthermore, they are executable: efficient declarative (and non-declarative) implementations (called attribute evaluators) are automatically derived by using well-known AG techniques. Finally, incremental implementations of the specified tools can be automatically generated from an attribute grammar.

Despite these advantages, attribute grammars are not of general use as a language-based tool specification formalism. In our opinion, this is due to two main reasons: firstly, there is no efficient, clear and elegant support for a component-based style of programming within the attribute grammar formalism. Although an efficient form of modularity can be achieved in AGs when each semantic domain is encapsulated in a single AG component [GG84,LJPR93,KW94] [CDPR98,SS99b,dMBS00], the fact is that there is no efficient support within the AG formalism for an easy reuse of such components. That is, how can a grammar writer "*plug in*" an AG component into her/his specification? How are those AG components *glued* together? How is information passed between different AG components? How can the separate analysis and compilation of components be achieved? Obviously we wish to provide answers to these questions within the attribute grammar formalism itself. Secondly, there is a lack of good generic, reusable attribute grammar components that can be easily "*plugged in*" into the specifications of language-based tools. Components that are themselves written in the AG formalism.

The purpose of this paper is two-fold: firstly, to propose a component-based style of programming in the (higher-order) attribute grammar formalism. This means that attribute grammar components are efficiently and easily "plugged-into" an AG specification via higher-order attributes. In this approach, one AG component defines a higher-order attribute which is decorated according to the attribute equations defined by another AG component.

Secondly, to introduce two generic, reusable and off-the-shelf AG components. These components are themselves defined in the HAG formalism and provide modern and powerful properties to visualize, animate and interact with language-based tools.

This paper is organized as follows: Section 2 presents higher-order attribute grammars, its notation and provides a simple example that will be used throughout the paper. Section 3 introduces HAG component-based programming and presents two generic AG components: a visualization and animation component (Section 3.1) and graphical user interface component (Section 3.2). Section 4 discusses related work and Section 5 contains the conclusions.

2 Higher-Order Attribute Grammars

The techniques presented in this paper are based on the *higher-order attribute grammar* formalism [VSK89]. Higher-Order Attribute Grammars are an important extension to the attribute grammar formalism. Conventional attribute grammars are augmented with *higher-order attributes*, the so-called *attributable attributes*. Higher-order attributes are attributes whose value is a tree. We may associate, once again, attributes with such a tree. Attributes of these so-called *higher-order trees*, may be higher-order attributes again. Higher-order attribute grammars have four main characteristics:

– First, when a computation can not be easily expressed in terms of the inductive structure of the underlying tree, a better suited structure can be computed before. Consider, for example, a language where the abstract grammar does not match the concrete one. Consider also that the semantic rules of such a language are easily expressed over the abstract grammar rather than over the concrete one. The mapping between both grammars can be specified within the higher-order attribute grammar formalism: the attribute equations of the concrete grammar define a higher-order attribute representing the abstract grammar. As a result, the decoration of a concrete syntax tree constructs a higher-order tree: the abstract syntax tree. The attribute equations of the abstract grammar define the semantics of the language.
– Second, semantic functions are redundant. In higher-order attribute grammars every computation can be modelled through attribution rules. More specifically, inductive semantic functions can be replaced by higher-order attributes. For example, a typical application of higher-order attributes is to model the (recursive) lookup function in an environment. Consequently, there is no need to have a different notation (or language) to define semantic functions in AGs. Moreover, because we express inductive functions by attributes and attribute equations, the termination of such functions is statically checked by standard AG techniques (*e.g.*, the circularity test).
– The third characteristic is that part of the abstract tree can be used directly as a value within a semantic equation. That is, grammar symbols can be moved from the syntactic domain to the semantic domain.
– Finally, as we will describe in this paper, attribute grammar components can be "glued" via higher-order attributes.

These characteristics make higher-order attribute grammars particularly suitable to model language-based tools [TC90,Pen94,KS98,Sar99].

2.1 The Block Language

Consider a very simple language that deals with the scope rules of a block structured language: a definition of an identifier `x` is visible in the smallest enclosing block, with the exception of local blocks that also contain a definition of `x`. In the latter case, the definition of `x` in the local scope hides the definition in the global one.

We shall analyse these scope rules via our favorite (toy) language: the BLOCK language[1]. One sentence in BLOCK consists of a *block*, and a block is a (possibly empty) list of *statements*. A statement is one of the following three things: a *declaration* of an identifier (such as `decl` *a*), the *use* of an identifier (such as `use` *a*), or a nested *block*. Statements are separated by the punctuation symbol

[1] The BLOCK language, that we introduced in [SSK97,Sar99], has become a popular example to study the static scheduling of "circular" definitions [dMPJvW99,Law01]

“;” and blocks are surrounded by square brackets. A concrete sentence in this language looks as follows:

```
sentence = [ use x ; use y ; decl x ;
             [ decl y ; use y ; use w ] ;
             decl y ; decl x
           ]
```

This language does not require that declarations of identifiers occur before their first use. Note that this is the case in the first two applied occurrences of `x` and `y`: they refer to their (latter) definitions on the outermost block. Note also that the local block defines a second identifier `y`. Consequently, the second applied occurrence of `y` (in the local block) refers to the inner definition and not to the outer definition. In a block, however, an identifier may be declared once, at the most. So, the second definition of identifier `x` in the outermost block is invalid. Furthermore, the BLOCK language requires that only defined identifiers may be used. As a result, the applied occurrence of `w` in the local block is invalid, since `w` has no binding occurrence at all.

We aim to develop a program that analyses BLOCK programs and computes a list containing the identifiers which do not obey to the rules of the language. In order to make the problem more interesting, and also to make it easier to detect which identifiers are being incorrectly used in a BLOCK program, we require that the list of invalid identifiers follows the sequential structure of the input program. Thus, the semantic meaning of processing the example sentence is `[w,x]`.

The BLOCK language does not force a *declare-before-use* discipline. Consequently, a conventional implementation of the required analysis naturally leads to a program that traverses each block twice: once for processing the declarations of identifiers and constructing an environment and a second time to process the uses of identifiers (using the computed environment) in order to check for the use of non-declared identifiers. The uniqueness of identifiers is checked in the first traversal: for each newly encountered identifier declaration it is checked whether that identifier has already been declared at the same lexical level. In this case, the identifier has to be added to a list reporting the detected errors. The straightforward algorithm to implement the BLOCK processor looks as follows:

1st Traversal	2nd Traversal
- *Collect the list of local definitions*	- *Use the list of definitions as the global environment*
- *Detect duplicate definitions (using the collected definitions)*	- *Detect use of non defined names*
	- *Combine “both” errors*

As a consequence, semantic errors resulting from duplicated definitions are computed during the first traversal, and errors resulting from missing declarations, in the second one. Thus, a “gluing” data structure has to pass explicitly the detected errors from the first to the second traversal, in order to compute the final list of errors in the desired order.

2.2 The Attribute Grammar for the Block Language

In this section we shall describe the program `block` in the traditional attribute grammar paradigm. To define the structure of the BLOCK language, we start by introducing one context-free grammar defining the abstract structure of Block. Then, we extend this grammar with attributes and the attribution rules.

We associate an inherited attribute *dcli* of type *Env* to the non-terminal symbols *Its* and *It* that define a block. The inherited environment is threaded through the block in order to accumulate the local definitions and in this way synthesizes the total environment of the block. To distinguish between the same identifier declared at different levels, we use an attribute *lev* that distributes the block's level. We associate a synthesized attribute *dclo* to the non-terminal symbols *Its* and *It*, which defines the newly computed environment. The total environment of a block is passed downwards to its body in the attribute *env* in order to detect applied occurrences of undefined identifiers. Every block inherits the environment of its outer block. The exception is the outermost block: it inherits an empty environment. To synthesize the list of errors we associate the attribute *errs* to *Its* and *It*.

The static semantics of the BLOCK language are defined in the attribute grammar presented in Fragment 1. We use a *standard* AG notation: productions are labelled for future references. Within the attribution rules of a production, different occurrences of the same symbol are denoted by distinct subscripts. Inherited (synthesized) attributes are prefixed with the down (up) arrow $\downarrow$ ($\uparrow$). Pseudo terminal symbols are syntactically referenced in the AG, *i.e.*, they are used directly as values in the attribution rules. The attribution rules are written as HASKELL-like expressions. Copy rules are included in the AG specification (although there are well-known techniques to omit copy rules, in this paper, we prefer to explicitly define them). The semantic functions *mBIn* (standing for "must be in") and *mNBIn* ("must not be in") define usual lookup operations[2].

$$\begin{array}{l}
Its < \downarrow lev : Int, \downarrow dcli : Env, \downarrow env : Env \\
\quad , \uparrow dclo : Env, \uparrow errs : Err > \\
Its = \mathsf{NilIts} \\
\quad Its.dclo = Its.dcli \\
\quad Its.errs = [\,] \\
\quad \mid\ \mathsf{ConsIts}\ It\ \ Its \\
\quad It.dcli \quad = Its_1.dcli \\
\quad Its_2.env \ = Its_1.env \\
\quad It.env \quad = Its_1.env \\
\quad Its_2.dcli = It.dclo \\
\quad Its_1.dclo = Its_2.dclo \\
\quad It.lev \quad\ = Its_1.lev \\
\quad Its_2.lev \ \ = Its_1.lev \\
\quad Its_1.errs = It.errs \mathbin{+\!+} Its_2.errs
\end{array}$$

$$\begin{array}{l}
It\ < \downarrow lev : Int, \downarrow dcli : Env, \downarrow env : Env \\
\quad , \uparrow dclo : Env, \uparrow errs : Err > \\
It = \mathsf{Use}\ \ \mathtt{String} \\
\quad It.dclo = It.dcli \\
\quad It.errs = mBIn\ (\mathtt{String}, It.env) \\
\quad \mid\ \mathsf{Decl}\ \ \mathtt{String} \\
\quad It.dclo = (\mathsf{Pair}\ \mathtt{String}\ It.lev) : It.dcli \\
\quad It.errs = mNBIn\ (\mathsf{Pair}\ \mathtt{String}\ It.lev, It.dcli) \\
\quad \mid\ \mathsf{Block}\ Its \\
\quad It.dclo \ = It.dcli \\
\quad Its.dcli = It.env \\
\quad Its.lev \ \ = It.lev + 1 \\
\quad Its.env = Its.dclo \\
\quad It.errs \ = Its.errs
\end{array}$$

Fragment 1: The BLOCK attribute grammar.

[2] These inductive functions can be defined via higher-order attributes. Indeed, in the BLOCK HAG presented in [Sar99], we have such an example.

It is common practice in attribute grammars to use additional non-terminals and productions to define new data types and constructor types, respectively. The type *Env* and the constructor function Pair are examples of that:

```
Tuple = Pair      String  Int
Env   = ConsEnv Tuple  Env
      | NilEnv
Err   = ConsErr  String  Err
      | NilErr
```

Note that, the type *Env* is isomorphic with non-terminal *Env*: the term constructor functions ConsEnv and NilEnv correspond to the HASKELL built-in list constructor functions `:` and `[]`, respectively. Roughly speaking, non-terminals define tree type constructors and productions define value type constructors. We will use both notations to define and to construct value types.

To make the AG more readable, we introduce a root non-terminal so that we can easily write the attribution rules specifying that the initial environment of the outermost block is empty (*i.e.*, the root is context-free) and that its lexical level is 0.

```
P < ↑ errs : Err >
P = Root Its
   Its.dcli = []
   Its.lev  = 0
   Its.env = Its.dclo
   P.errs  = Its.errs
```

The above fragment includes a typical equation where a inherited attribute (*env*) depends on a synthesized attribute (*dclo*) of the same non-terminal (*Its*). Although such dependencies are natural in attribute grammars they may lead to complex and counterintuitive solutions in other paradigms (functional, imperative, etc), because they induce additional traversal functions which have to be explicitly “glued” together to convey information between them.

The AG fragments presented so far formally specify the static semantics of the BLOCK language. A higher-order extension to this AG will be presented in next section, where we introduce our component-base programming techniques.

3 Gluing Grammar Components via Higher-Order Attribute Grammars

In functional programming, it is common practice to use intermediate data structures to convey information between functions. One function constructs the intermediate data structure which is destructed by another one. The intermediate data structure is the component “glue”. We will mimic this approach in the higher-order attribute grammar setting: an AG component defines (or, at attribute evaluation time, constructs) a higher-order attribute (*i.e.*, a tree-like data structure), which is used (or decorated) by the other AG component.

This gluing of AG components is defined in the HAG formalism itself as follows: consider, for example, that an AG component, say AG_{reuse}, expresses some algorithm $\mathcal{A}$ over a grammar rooted X, and suppose that we wish to express the same algorithm when defining a new grammar, say AG_{new}. Under the higher-order formalism this is done as follows: firstly, we define an attributable attribute, say a with type X, in the productions, say P, of AG_{new} where we need to express algorithm $\mathcal{A}$. Secondly, we extend AG_{new} with attributes, whose types are the types (*i.e.*, non-terminals) defined in AG_{reuse}, and attribute equations, where the semantic functions are the constructors (*i.e.*productions) of AG_{reuse}. That is, we define attributes that are tree-value attributes. After that, we instantiate the higher-order attribute a with the tree-value attribute of type X constructed in the context of production P. Then, we instantiate the inherited attributes of associated type/non-terminal (*i.e.*, X). Finally, and by definition of HAGs, the generated synthesized attribute occurrences of a are defined by the attribute equations of AG_{reuse}. They are ready to be used in the attribute grammar specification, like any other first-order attribute.

Notice that by expressing the gluing of AG components within the AG formalism itself, we are able to use all the standard attribute grammar techniques, *e.g.*, the efficient scheduling of computations and the static detection of circularities. For example, the inherited/synthesized attributes of the AG components can be "connected" in any order. The HAG writer does not have to be concerned with the existence of cyclic dependencies among AG components: the AG circularity test will detect them for him. Furthermore, we can use attribute grammar techniques to derive efficient implementations for the resulting HAG. For example, we can use our deforestation techniques to eliminate the possibly large intermediate trees that glue the different components [SS99a].

Most of the powerful attribute grammar techniques are based on a global static analysis of attribute dependencies. Thus, they require that the different AG modules/components are "fused" into an equivalent monolithic HAG, before they are analised. In [SS99b] we have presented techniques to achieve the separate analysis and compilation of AG modules than naturally extend to our component-based approach.

3.1 An Attribute Grammar Component for Visualization and Animation of Language-based Tools

In order to be more precise about our approach, let us consider the BLOCK language example again. Because this simple toy example has a non-trivial scheduling of computations, we would like to "plug into" the AG specification an AG component that allows us to visualize and animate the BLOCK processor.

Thus, we introduce a generic component for the visualization and animation of AGs. We wish to use this AG as a *generic visual and animation AG component*. We start by defining an abstract grammar that is sufficiently generic to define all possible abstract tree structures we may want to visualize and animate. The grammar is as follows:

```
TreeViz   = CTreeViz       TreeId  [TreeStmt]

TreeStmt  = CStmtNode      NodeStmt
          | CStmtEdge      EdgeStmt
          | CStmtAttr      AttrStmt
NodeStmt  = CNodeStmt      NodeId  [Attr]
EdgeStmt  = CEdgeStmt      NodeId  [EdgeRHS]  Attrs
EdgeRHS   = CRHSExpNode    EdgeOp  NodeId
Attr      = CAttr          AttrId  AttrVal
```

The non-terminals *TreeId*, *NodeId*, *EdgeOp*, *AttrId*, *AttrVal* define sequences of characters (strings). In order to make it easier to use this component, we define a set of functions/macros that, using the productions of this AG component, define usual occurring node formats in our trees. Next, we present four functions that define the shape of a node as a record (*attrShapeRecord*), as a circle (*attrShapeCircle*), as the value of a node label (*attrLabel*), and, finally, as a node that contains a value and an arrow to a child node. These functions are presented next.

```
attrShapeRecord               = CAttr "shape" "record"
attrShapeCircle               = CAttr "shape" "circle"
attrLabel label               = CAttr "label" label
nodeRecord1 val father child =
  [CStmtNode (CNodeStmt father) [attrShapeRecord , attrLabel (val ++ "|<c>")]
  ,CStmtEdge (CEdgeStmt "c") [CRHSExpNode "->" child] ]
```

The label is a string that defines the format of the node record. The non-terminal *EdgeOp* is a string defining the direction of the arrow.

The above grammar defines the abstract structure of abstract trees only. To have a concrete graphical representation of the trees, however, we need to map such abstract tree representation into a concrete one. Rather than defining a concrete interface from scratch and implementing a tree/graph visualization system (and reinventing the wheel!), we can synthesize a concrete interface for existing high quality graph visualization systems, *e.g.*, the GraphViz system [GN99]. We omit here the attributes and attribution rules that we have associated to the visualization grammar since they are neither relevant to reuse this component nor to understand our techniques.

To reuse this component, however, we need to know the inherited and synthesized attributes of its root non-terminal, *i.e.*, the *interface* of the AG component. This grammar component is context-free (it does not have any inherited attributes) and synthesizes two attributes *graphviz* and *xml*, both of type string. These two attributes synthesize a textual representation of trees in the GraphViz input language. The first attribute displays trees in the usual graphic tree representation, while the second one uses a Xml tree-like representation (where the production names are the element tags).

$TreeViz < \uparrow graphviz : String, \uparrow xml : String >$

We are now in position to "glue" this component to the BLOCK AG. Let us start by defining the attribute and the equations that specify the construction of the GraphViz representation.

```
Its < ↑ viztree : [TreeStmt] >
Its = NilIts
    Its.viztree = nodeEmptyCircle  treeRef(Its)
    | ConsIts It  Its
    Its_1.viztree = (nodeRecord2  ""  treeRef(Its_1)  treeRef(It)  treeRef(Its_2))
                    ++ It.viztree ++ Its_2.viztree
It  < ↑ viztree : [TreeStmt] >
It  = Use    String
    It.viztree = nodeRecord0  ("Use" ++ String)  treeRef(It)
    | Decl   String
    It.viztree = nodeRecord0  ("Decl" ++ String)  treeRef(It)
    | Block  Its
    It.viztree = (nodeRecord1  "Block"  treeRef(It)  treeRef(Its)) ++ Its.viztree
```

Fragment 2: Constructing the Visual Tree.

Where the function *treeRef* returns a unique identifier of its tree-value argument (the tree pointer).

Next, we declare a higher-order attribute, *i.e.*, attributable attribute (*ata*) named *visualTree*, in the context of the single production applied to the root non-terminal of the BLOCK AG. The type of the higher-order attribute is *TreeViz* which is the type of the root non-terminal of the reused component. After that, we have to instantiate the higher-order attribute with the attribute synthesized in the above fragment. Finally, and because *TreeViz* has no inherited attributes, we just have to access the synthesized attribute of the higher-order attribute, as usual. The HAG fragment looks has follows:

```
P < ↑ String : visualTree >
P = Root Its
   ata  visualTree : TreeViz                                -- Declaration
   visualTree  = CTreeViz  "BlockTree"  Its.viztree         -- Instantiation
   P.visualTree = visualTree.graphviz                       -- Use of its syn. attrs
```

This fragment defines a higher-order extension to the BLOCK attribute grammar presented in the previous section. To process such higher-order attribute grammar, we use the LRC system: an incremental, purely functional higher-order attribute grammar based system [KS98]. Thus, we can use LRC to process the BLOCK HAG and to produce the desire BLOCK processor.

Figure 1 shows two different snapshots (displayed by GraphViz) of the tree that is obtained as the result of running the BLOCK processor with the input example sentence. As we can see the tree is collapsed into a minimal *Direct*

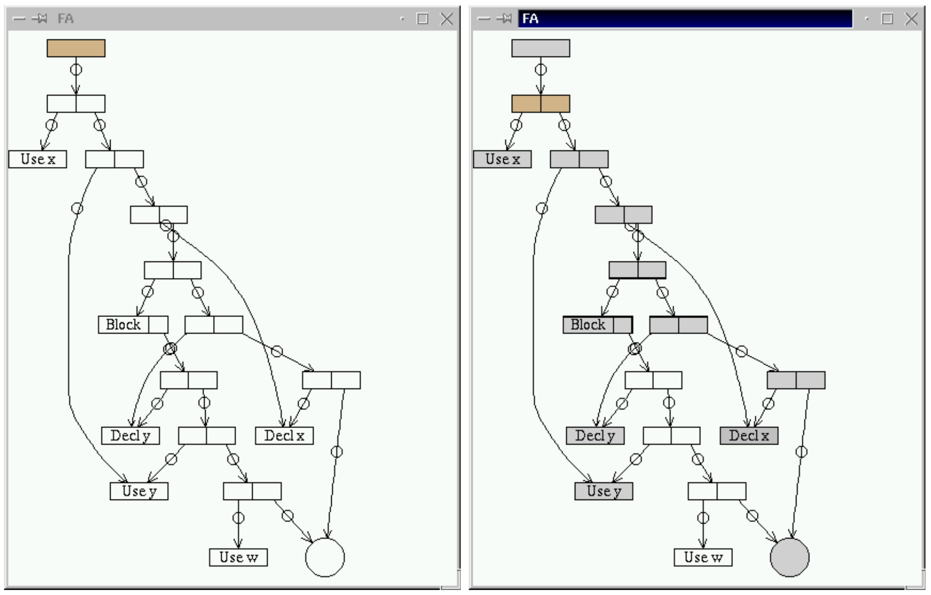


Fig. 1. The DAG representing the BLOCK example sentence at the beginning of the evaluation (left) and after completing the first traversal to the outermost block (right).

Acyclic Graphs (DAG). This happens because we are using the incremental model of attribute evaluation of LRC[3].

Besides computing the graphical representation of the tree, the processor generated by LRC also produces a sequence of node transitions. This is exactly the sequence of visits the evaluator performs to decorate the tree under consideration. Such sequence can be loaded in and animated in GraphViz, either in single step or in continuous mode, forwards and backwards. Furthermore, colors are used to mark the visited nodes.

The snapshot on the left shows the beginning of the evaluation: the root node is visited for the first time (the shadowed node). The snapshot on the right shows the end of the first traversal to the outermost block. Note that the nodes of the nested block were not visited (they are not shadowed). Indeed, the AG scheduler induced (as we expected) that only after collecting the complete environment of the outer block (performed on its first traversal), can the evaluator visit the inner ones. The inner blocks are traversed twice in the second traversal of the outer block.

[3] LRC achieves incremental evaluation through function memoization. Trees are arguments of the evaluators' functions. Thus, to make function memoization possible, they have to be efficiently compared for equality. Minimal DAG's allow for efficient equality tests between all terms because a pointer comparison suffices.

3.2 An Attribute Grammar Component for Advanced Interactive Interfaces

As it was previously stated, types can be defined within the attribute grammar formalism. So, we may use this approach to introduce a type that defines an abstract representation of the interface of language-based tools. In other words, we use an abstract grammar to define an abstract interface. The productions of such a grammar represent "standard" graphical user interface objects, like menus, buttons, etc. Next, we present the so-called *abstract interface grammar*.

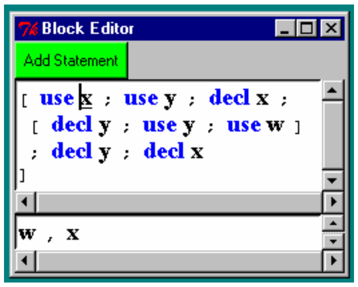
```
Visuals = CVisuals [Toplevel]                 Frame = Label          String
                                                    | ListBox        Entrylist
                                                    | PullDownMenu   String MenuList
Toplevel = Toplevel Frame  String  String           | PushButton     String
                                                    | Unparse        Ptr
                                                    | HList          [Frame]
                                                    | VList          [Frame]
```

The non-terminal *Visual* defines the type of the abstract interface of the tool: it is a list of Toplevel objects, that may be displayed in different windows. A Toplevel construct displays a frame in a window. It has three arguments: the frame, a name (for future references) and the window title. The productions applied to non-terminal *Frame* define concrete visual objects. For example, production PushButton represents a *push-button*, production ListBox represents a *list box*, etc.

The production Unparse represents a visual object that provides *structured text editing* [RT89]. It displays a pretty-printed version of its (tree) argument and allows the user to interact with it. Such beautified textual representation of the abstract syntax tree is produced according to the unparse rules specified in the grammar. It also allows the user to point to the textual representation to edit it (via the keyboard), or to transform it using user defined transformations. The productions VList and HList define combinators: they vertically and horizontally (respectively) combine visual objects into more complicated ones. These non-terminals and productions can be directly used in the attribute grammar to define the interface of the environments. Thus, the interface is specified through attribution, *i.e.*, within the AG formalism.

To define a concrete interface, we need, as we have said above, to define the mapping from the abstract interface representation into a concrete one. Instead of defining a concrete interface from scratch, we synthesize a concrete interface for a existing GUI toolkit, *e.g.*, the TCL/TK GUI toolkit [Ous94]. Indeed, the GUI AG component synthesizes TCL/TK code defining the interface in the attribute named *tk*.

Next, we present an attribute grammar fragment that glues the BLOCK HAG with this GUI AG component. It defines an interactive interface consisting of three visual objects that are vertically combined, namely: a push-button, the unparsing of the input under consideration and the unparsing of the list of errors. The root symbol *P* synthesizes the TCL/TK concrete code in the attribute occurrence *concreteInterface*.



```
[ use x ; use y ; decl x ;
 [ decl y ; use y ; use w ]
 ; decl y ; decl x
]
```

```
w , x
```

Fig. 2. The BLOCK environment's interface generated from the HAG.

```
P < ↑ concreteInterface : Tk >
P = Root Its
   ata  absInterface : Visuals
   absInterface          = let { button =  PushButton  "Add Statement"
                                 editor  =  Unparse  &P
                                 errors =  Unparse  &P.errs
                                 comb    =  VList  [ button , editor , errors ]
                               } in [ Toplevel  comb "edit" "Block Editor" ]
   P.concreteInterface = absInterface.tk
```

Fragment 3: The BLOCK graphical user interface.

Figure 2 shows the concrete interface of the BLOCK processor.

The PushButton constructor simply displays a push-button. To assign an *action* to the displayed button we have to define such an action. Once again we use the same technique, *i.e.*, we define an abstract grammar to describe the abstract events handled by interactive interfaces. Basically, we associate an abstract event-handler to each visual object.

```
Event = ButtonPress   String
      | ListBoxSelect Entrylist
      | MenuSelect    String
      | TextKeyPress  Char
```

The constructor ButtonPress is the event-handler associated with PushButton. Next, we show a possible action associated with this event-handler.

```
Its = NilIts
    bind on ButtonPress  "Add Statement"
        :   Its  →  ConsIts (Decl "a")  NilIts;
```

The *bind* expression is used to specify how user interactions are handled by the language-based environment. In this case, it simply defines that every time the push-button `"Add Statement"` is pressed, the rooted subtree *Its* is transformed into ConsIts Decl("a") NilIts. Note that this event-handler constructor is defined in the context of a NilIts production. Thus, a new declaration is added at the end of the program being edited.

Other features of visualization and animation, and of the advanced graphical user interface AG components are:

– The use of abstract grammars (*i.e.*, intermediate representation languages) makes these components highly modular: new concrete visualizations/animations/interfaces can be "*plugged into*" the AG system, just by defining the corresponding mapping function.
– This approach has another important property: under an incremental attribute evaluation scheme, *the visualization/animation/interface is incrementally computed*, like any other attribute value [Sar99,SSK00].
– Because the Lrc system uses an incremental computational model, we can animate incremental attribute evaluators. Indeed, in the animations produced by Lrc, it is possible to visualize the reuse of a memoized function call: the animation simply changes the color of a node, without visiting its descendents.

4 Related Work

The work presented in this paper is closely related to attribute coupled grammars [GG84,LJPR93,CDPR98], composable attribute grammars [FMY92] and Kastens and Waite work on modularity and reusability of attribute grammars [KW94].

Attribute coupled grammars consist of a set of AG components each of which (conceptually) returns a tree-valued result that is the input for the next component. Grammars are coupled by defining attribute equations that build the required tree-valued attributes, very much like the values of higher-order attributes are defined in our approach (*e.g.*, Fragment 2). In attribute coupled grammars, however, the flow of data is strictly linear and unidirectional. In our approach the data can flow freely throughout the components, provided that no attribute depends directly nor indirectly on itself. Under our techniques such cyclic dependencies are statically detected.

In [GG84] descriptional composition is defined to eliminate the creation of the intermediate trees. That is, from the coupling attribute grammar (modules) a grammar is constructed that defines the same equations, but that eliminates the construction of the intermediate trees. The descriptional composition, however, can result in a non-absolute circular AG. Furthermore, descriptional composition does not allow the separate analysis and compilation of grammar components.

Composable attribute grammars [FMY92] use a particular grammar module for gluing AG components. Grammar modules con be analised and compiled separately. However, the gluing of the components is expressed with a special notation outside the AG formalism.

Kastens and Waite [KW94] aim at a different form of modularity. They show that a combination of notational concepts can be used to create reusable attribution modules. They also define a set of modules to express common operation on programming languages. However, such modules are not defined within the

AG formalism, thus, making the maintenance, updating and understanding of such components much harder.

5 Conclusions

This paper presented techniques to write attribute grammars under a component-based style of programming. Such techniques rely entirely on the higher-order attribute grammar formalism: attribute grammar components are glued into a larger AG system through higher-order attributes. Standard attribute grammar techniques are used to detect circularities (*e.g.*, AG circularity test), to efficiently schedule the computations (*e.g.*, AG scheduling algorithms), and, to eliminate redundant intermediate data structures induced by higher-order attributes (*e.g.*, AG deforestation techniques).

We also have presented two generic, reusable and off-the-shelf AG components that can easily be "plugged into" any higher-order attribute grammar specification. Such components provide powerful properties to visualize, animate and interact with language-based tools. Thanks to the fact that these components are themselves defined in the HAG formalism, we inherit all of its nice properties and because of that the maintenance, updating and understanding of such components is simpler.

These components are implemented in the Lrc system. However, they can be reused in any attribute grammar system, provided it processes higher-order attribute grammars.

References

[CDPR98] Loic Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Generic Programming by Program Composition. In *Proceedings of the Workshop on Generic Programming*, pages 1–13, June 1998.

[dMBS00] Oege de Moor, Kevin Backhouse, and Doaitse Swierstra. First-Class Attribute Grammars. In D. Parigot and M. Mernik, editors, *Third Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 1–20, Ponte de Lima, Portugal, July 2000. INRIA Rocquencourt.

[dMPJvW99] Oege de Moor, Simon Peyton-Jones, and Eric van Wyk. Aspect-Oriented Compilers. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, LNCS, September 1999.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute Coupled Grammars. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19, pages 157–170, Montréal, June 1984.

[GN99] Emden R. Gransner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 00(S1):1–29, 1999.

[KS98] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

[KW94] Uwe Kastens and William Waite. Modularity and reusability in attribute grammar. *Acta Informatica*, 31:601–627, June 1994.

[Law01] Julia L. Lawall. Implementing Circularity Using Partial Evaluation. In *Proceedings of the Second Symposium on Programs as Data Objects PADO II*, volume 2053 of *LNCS*, May 2001.

[LJPR93] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *LNCS*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

[Ous94] J.K. Ousterhout. *Tcl and the Tk toolkit.* Addison Wesley, 1994.

[Pen94] Maarten Pennings. *Generating Incremental Evaluators.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 1994. `ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Pennings/`.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator.* Springer, 1989.

[Sar99] João Saraiva. *Purely Functional Implementation of Attribute Grammars.* PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999. `ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

[SS99a] João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

[SS99b] João Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[SSK97] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Strictification of Computations on Trees. Technical report UU-CS-1997-30, Department of Computer Science, Utrecht University, August 1997. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1997/1997-30.ps.gz`.

[SSK00] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. In David Watt, editor, *9th International Conference on Compiler Construction, CC/ETAPS2000*, volume 1781 of *LNCS*, pages 279–294. Springer-Verlag, March 2000.

[TC90] Tim Teitelbaum and Richard Chapman. Higher-order attribute grammars and editing environments. In *ACM SIGPLAN'90 Conference on Principles of Programming Languages*, volume 25, pages 197–208. ACM, June 1990.

[VSK89] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. Higher order attribute grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24, pages 131–145. ACM, July 1989.